Pynamic: the Python Dynamic Benchmark

Gregory L. Lee, Dong H. Ahn, Bronis R. de Supinski, John Gyllenhaal, Patrick Miller*

Lawrence Livermore National Laboratory, Livermore, CA, USA

{lee218, ahn1, bronis, gyllen, miller220}@llnl.gov

.

Abstract

Python is widely used in scientific computing to facilitate application development and to support features such as computational steering. Making full use of some of Python's popular features, which improve programmer productivity, leads to applications that access extremely high numbers of dynamically linked libraries (DLLs). As a result, some important Python-based applications severely stress a system's dynamic linking and loading capabilities and also cause significant difficulties for most development environment tools, such as debuggers. Furthermore, using the Python paradigm for large scale MPI-based applications can create significant file IO and further stress tools and operating systems. In this paper, we present Pynamic, the first benchmark program to support configurable emulation of a wide-range of the DLL usage of Python-based applications for large scale systems. Pynamic has already been used to accurately reproduce system software and tool issues encountered by important large python-based scientific applications on our supercomputers. Pynamic provided insight for our system software and tool vendors, and our application developers, into the impact of several design decisions. As we describe the Pynamic benchmark, we will highlight some of the issues discovered in our large scale system software and tools using Pynamic.

1 Introduction

Python is a widely used, interpreted, object-oriented programming language. Python's standard library includes many high-level data types and advanced functionality. Interface generators like SWIG [4] can wrap existing C and C++ code for use within Python. Frequently, users write modules in C or C++ to extend Python beyond its base functionality. Many open-source modules exist for numeric and scientific computing, GUI development, web programming, and parallel computing. These properties make Python a popular choice for rapid prototyping, for application steering, and for developing complex multipurpose codes.

While Python is highly portable, its dynamic nature can stress operating systems and development environment tools such as debuggers. Python modules are implemented as dynamically linked libraries (DLLs). Since DLL symbol and address resolution are performed at runtime, either applications must use few DLLs or the OS and file systems must efficiently support this process. Further, development environment tools must be aware of and track this dynamic process, which requires efficient mechanisms to handle runtime changes to the executable code.

For large applications, the number of DLLs can extend into the hundreds. When validating a new system or measuring tool performance, porting all of these modules may be very time consuming or may just not be practical. With some applications, access and licensing restrictions may even prevent distributing all of the code to third party vendors. To overcome these limitations, we have developed Pynamic, a customizable Python dynamic library benchmark based on pyMPI [11], a Python extension that provides access to the MPI communication library. Pynamic generates a configurable number of Python modules and utility libraries that can be used to match the footprint of actual Python-based codes. Pynamic includes the configuration settings required to match the footprint of one of our important large-scale python applications; this configuration stresses systems and tools. Thus, Pynamic supports accurate testing of system aspects stressed by the full range of Python applications, particularly for large scale systems. To the best of our knowledge, no existing benchmark provides Pynamic's breadth of stress testing.

This paper presents details of the Pynamic benchmark. We first discuss Python and the system aspects that Python applications stress in Section 2. Next, we present details of Pynamic's implementation in Section 3. Section 4 then presents initial results with Pynamic that demonstrate how it can help guide OS and tool development. Finally, our conclusion in Section 5 discusses possible system software changes that Pynamic might help motivate for large scale systems.

[*] currently of D.E. Shaw Research, LLC

[0] This work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (UCRL-ABS-230099)

2 Python and pyMPI

Python is a flexible scripting language used for a wide variety of purposes from web applets to console games [3]. Even though applications written purely in scripted languages may run hundreds of times slower than their compiled counterparts, the flexibility and low total user time to solution makes them attractive. Oftentimes a hybrid approach is used to bridge the performance gap. In this case, the scripting language plays the role of coordinator. Intensive computing is done in *modules* written in a compiled language like C, C++, or FORTRAN. Wrapper generation tools like SWIG and F2PY [12] simplify creating Python modules from compiled source code.

pyMPI was developed to extend Python's scripting abilities to parallel and distributed codes. It established an MPI environment in which parallel extension modules are written. Objects within these modules communicate internally with MPI. The pyMPI tool launches a Python interpreter for each MPI process with separate communicators. The pyMPI processes can themselves send messages using MPI-like semantics. pyMPI handles the details of serializing/unserializing messages using MPI native types where possible and the Python pickle serialization mechanism elsewhere. This allows coordination code to be written with parallel awareness (e.g., selecting the minimum timestep with mpi.allreduce(dt,mpi.MIN)).

2.1 Scientific Computing with Python

In the mid-1990's a number of groups discovered that they could adapt Python to scientific scripting needs. Modules such as *Numeric* started to be developed that made Python useful in a scientific context. Soon, entire computations could be performed completely within the Python interpreter. Even large scale parallel computations could be performed in this way [5].

One of the key advantages that Python offers is the sandbox effect. Modules and processing can be combined in one convenient place to simplify processing. A single Python script can provide setup, simulation, instrumentation, and postprocessing. Synchronizing these activities through extant memory instead of through the file system is more efficient and simpler. Rather than have separate tools that process input files to output files, a Python script imports several modules that pass references to data. This ideal scientific exploration programming environment encouraged the creation of general tools like the SciPy toolkit [10]. This toolkit provides visualization, optimization, statistics, linear algebra, and other handy tools that simplify writing scientific codes.

2.2 OS, Runtime Systems, and Development Tools Consideration

Inarguably, Python's flexibility provides many benefits to the high performance computing (HPC) community. The adoption of this programming language, however, has imposed a new class of workloads on several key system software components. Python-based scientific applications exercise and often stress these components in a fashion distinguished from more traditional HPC programs. Thus, they expose certain weak links in the HPC system software stack. These system aspects may fail to work at all or may perform so poorly as to make the Python-based paradigm unusable. Specifically, the heavy use of dynamic linking and loading can stress the system, from its OS and runtime system layers to the development environment, including debuggers and performance analysis tools. Larger applications, in terms of code size and number of DLLs, and larger jobs, in terms of node counts, prove particularly difficult. Thus, it is clear that we must address these issues for future extreme scale machines.

We expect this emerging class of applications will continue to grow in ways that employ an ever increasing number of dynamic libraries. More functionality will be added in a compartmentalized manner and jobs will run on an ever increasing number of nodes. Thus, we need Pynamic, a benchmark that easily captures current and future application profiles.

In the following, we consider several key system software components and sample problems, sensitive to these new workloads. Though not exhaustive, the software components and problems are the main parameters Pynamic is designed to capture in testing and stressing a system. We will also discuss some implications for future systems.

2.2.1 Dynamic Linking and Loading

Dynamic linking defers much of the linking procedure until a program starts. This deferral facilitates library updates without requiring all applications that use the updated library to be re-linked. More importantly, it enables dynamic loading and unloading of part of a program. Such benefits, however, come at the cost of performance overheads. To allow the runtime relocation of codes, indirection mechanisms such as IBM AIX's Table of Contents [8] or Excutable Linkage Format's Global Offset Table [1] are often employed, incurring extra overheads and ultimately producing slower codes. Besides the pure costs involved in hopping through the extra levels of indirection, some commonly used dynamic linking schemes such as lazy binding can increase cache conflict misses.

For traditional HPC programs that only use a handful of libraries, the costs of dynamic linking and loading do not outweigh their benefits. But these overheads can become substantial for Python-based applications, which use DLLs extensively. For example, one of Lawrence Livermore National Laboratory's (LLNL's) important production multiphysics applications uses nearly five hundred libraries. We have found that OS and runtime DLL support become significant performance factors at this scale; the sheer volume of DLLs is the issue, not the operations that they perform. Building such a large quantity of libraries becomes impractical for third-party system and tool vendors to do testing. Further, many of the libraries have access restrictions, such as export control. Thus, a benchmark that generically captures an application's DLL profile will allow system implementers to improve the performance of large scale dynamic linking and loading without having to deal with the additional complexity of the actual application.

2.2.2 Operating System

Similarly, Python-based HPC applications demand more scalability from OS services. Programmers frequently use Python to splice many interchangeable packages with different characteristics together since they can then choose which packages to use at runtime, even steering the application in response to observed initial results. Thus, many Python applications have a very large overall code size. These ever-increasing text sizes conflict with a hard limit on the text size, such as that of the AIX 32-bit process address space, which limits the text size to 256MB. This issue is exacerbated by the trend toward multicore systems in which the per-core memory becomes significantly reduced. Trading off a complex memory management scheme for simplicity (e.g., disabling demand paging, a trend in contemporary massively parallel systems [2]) must factor in the efficient memory management requirements for large text sizes.

On extreme scale systems, we must also consider IO system requirements. A common practice of staging the executable onto the NFS file system while having input data and output on a parallel file system may no longer prove viable for Python-based applications for extreme scale runs. Simply put, an NFS file system could not support the level of parallel accesses without OS extensions such as collective opening of DLLs.

On the other hand, we must examine the services that can adversely impact the scalability of the upper client software stack, notably the development environment tool chain. For instance, IBM AIX versions prior to 4.3.2 mandated a client of the *ptrace* interface to reinsert all existing breakpoints on each load or unload event. Such an interface requirement renders client tools inoperable on Python-based applications beyond a certain scale. Randomizing the loaded addresses of dynamic libraries, such as in RedHat's exec-shield scheme [9], can also significantly degrade the scalability of tools that must compute, maintain or even re-parse information to deal with the heterogeneous process link maps of a job. Generally, scalable tools require the application processes running on a massively parallel system to have as homogeneous characteristics as possible. We need benchmarks that easily expose where services break this assumption and, thus, significantly stress the development environment.

2.2.3 Development Tool Chain

Many development tools have performed poorly on Python based applications. This failure is due in part to functionality gaps in the development tools aimed at debugging or optimizing scripts and native binaries simultaneously. However, the massive dynamic linking and loading behaviors stress even state-of-the-arts tools, particularly ones that exploit process control interfaces such as *ptrace*. To function correctly, tools like the TotalView parallel debugger [13] or the DynInst [7] dynamic instrumentation library must be notified of every dynamic linking and loading event so that they can update their internal process representations with respect to the newly loaded images. The unprecedented number of dynamic linking and loading events produced by Python-based applications overwhelms even the best designed update mechanisms.

To demonstrate, consider an application that links and loads M libraries and runs at N MPI tasks. When running under tool control, the application tasks must stop and wait for the tool update mechanism at least M x N times. Thus, the cost is roughly M x N x T_1 where T_1 is the time to complete handling a single event. Even worse, those events tend to occur early in the execution, either during binding time or in the beginning of runtime, making the tool's startup time unbearable. Besides, the process control interface of an OS can impose other requirements on each dynamic linking and loading event like that of AIX mentioned in the above section. In such a system, the penalty becomes M x N x (T_1 + (B x T_2)) where B is the number of the existing breakpoints and T_2 is the time it takes to reinsert a breakpoint. Even on a medium size run, the total cost becomes ∼500(shared libraries) x ∼500 (tasks) x (∼10 msec + (∼10 (breakpoints) x ∼1 msec)) = ∼83 minutes! Having to reinsert breakpoints approximately doubles the already excessive ∼41.5 minutes required just to process M x N libraries. Clearly, we need benchmarks, like Pynamic, that support evaluation of tool optimizations that reduce this type of cost.

3 Pynamic Implementation

The heart of Pynamic is the shared object generator that creates Python modules, collections of C functions that can be called from Python. Extensions to Python are commonly written in C and existing C libraries can also be interfaced into Python using a wrapper generator such as SWIG. The code generated by Pynamic isn't designed to do any insightful computation; rather it tests a system's linking and loading capabilities.

When configuring Pynamic, the user specifies the number of modules to generate as well as the average number of functions per module. The actual number of functions will vary based on a random number; a seed value can be specified, allowing for reproducible results. The function signatures vary from zero to five arguments of standard C types (int, long, float, double, char *). Each module contains a single Python-callable entry function that visits all of the module's functions up to a specifiable maximum depth. Specifically, with the default maximum depth of ten, the entry function calls every tenth function within that module. Each function then calls the next function until a depth of ten is reached, at which point control is eventually returned back to the entry function. This call chaining is typical of Python-based applications.

Additional complexity can be added to the generated code. Many Python modules have dependencies on external libraries such as physics packages or math libraries. Pynamic can mimic these dependencies with the generation of utility libraries. The user can specify the number of utility libraries to generate as well as the average number of functions per library. These utility library functions will then be called at random by the Python module functions. In addition, some Python modules are further dependent on other Python modules. When enabled, Pynamic will also generate an additional function per module that can be called by other modules.

Pynamic also creates a Python driver script. This script imports all generated modules and executes each module's entry function. In the presence of pyMPI, the driver will also perform a test of the MPI functionality. In its default form, the Pynamic driver is simply a functionality test. When enabled, the Pynamic driver can also gather performance metrics including the job startup time, module import time, function visit time, and the MPI test time. These metrics can provide valuable information when comparing a DLL-linked pyMPI to a non-linked, vanilla pyMPI build and also when exploring various linking and loading options.

Pynamic supports several different build and run configurations. For example, the shared objects can be linked into pyMPI at compile time. Several real world codes do this in order to mitigate the runtime cost of dynamically loading a Python module during the *import* command. Alternatively, the Pynamic driver can be run with a vanilla pyMPI build, or in a non-parallel environment, with a standard Python build. In fact, running Pynamic both linked into pyMPI and with a vanilla pyMPI build can provide some insight into the performance benefits of linking at compile time, as we discuss in the following section.

4 Results

Pynamic's timing measurements can provide useful insight to OS, tool, and application developers. We performed tests on Zeus, a 288 node, Infiniband-connected cluster at LLNL. Each node has 4 dual core, 2.4 GHZ Opteron processors. Zeus runs the Clustered High Availability Operating System, an OS based on Red Hat Enterprise Linux with the addition of cluster management support. Performance numbers were gathered for a Pynamic build designed to model the executable properties of an important Python-based multiphysics application at LLNL. The application incorporates about 500 shared libraries, more than half of which (57 percent) are Python modules. To match it as closely as possible, we configured Pynamic with an average of 1850 functions per library.

4.1 Vanilla pyMPI vs. Linked pyMPI

Our first set of tests compares runs of a vanilla pyMPI build that imports the modules dynamically to a build with the generated libraries linked into pyMPI. We gathered the time to import all of the generated modules as well as the time to visit all of those modules' functions. Comparing the Vanilla row in Table 1 to the Link row shows that linking all of the shared objects into the pyMPI executable resulted in a three fold speedup of the module import time, due in part to link time address resolution and to lazy procedure binding.

pyMPI	time in seconds			
version	startup	import	visit	total
Vanilla	1.5	152.8	2.9	157.2
Link	5.7	56.4	269.4	331.5
Link+Bind	285.6	58.2	2.8	346.6

Table 1: Pynamic results

Table 1 also shows the execution time for visiting all of the modules' functions. These results show that linking the shared libraries into pyMPI results in a large performance degradation, 100 times slower than a vanilla pyMPI build visiting all of the same module functions. This result may seem counterintuitive: doing additional symbol and address resolution at link time had a negative effect on runtime performance. However, it appears that the difference arises from the dynamic linker's behavior with respect to resolving undefined symbols including the Global Offset Table (GOT) and Procedure Linkage Table (PLT). Specifically, the vanilla pyMPI version resolves both the GOT and PLT when the modules are imported as it passes the RTLD_NOW flag to the *dlopen* call. The linked pyMPI version, however, does not resolve the PLT at *import* time because *dlopen* does not respect the RTLD_NOW flag for the modules that have already been linked with lazy binding at program startup. Instead, the PLT is filled when the functions are first referenced.

To test this hypothesis, we reran the tests with the LD_BIND_NOW environment variable set, which causes the dynamic linker to resolve the PLT at program startup. We show the results from these tests in the Link+Bind row of Table 1 (the results for the vanilla build of pyMPI where unchanged by the environment variable setting). We also include startup time, the time between program invocation and the first line of code, which we measure roughly by sending the current time as a command line argument and then comparing that value to a timestamp gathered immediately by the Pynamic driver. The results clearly show that setting the LD_BIND_NOW variable shifts the time required to fill the PLT from function execution time to program startup time, thus confirming our hypothesis.

To examine the timing discrepencies further, we instrumented the code with the Performance Application Programming Interface (PAPI) [6], an API to gather hardware performance counter events. We were specifically interested in the number of L1 and L2 data and instruction cache misses. We needed to gather data at the Python level, but PAPI only has a C interface. To overcome this limitation, we implemented our PAPI function calls within a python callable module. This module was then interfaced by the Pynamic driver to get the cache miss counts for both importing the modules and visiting the module functions. The cache miss counts can be seen in Table 2.

pyMPI version	import misses		visit misses	
	L1-D	L1-I	L1-D	L1-I
Vanilla	6269.8	0.47	3.9	18.0
Link	4945.2	0.25	3076.5	19.8
Link+Bind	4945.3	0.26	3.9	17.9

Table 2: Millions of L1 data and instruction cache misses.

Combining the results shown in Table 1 and 2, we can infer several interesting characteristics of this system's linking and loading capabilities. First, there seems to be a general inefficiency in the LINUX *dlopen* implementation when it deals with pre-linked shared objects. The *dlopen* call is supposed to increase only the reference count for the requested shared object if it has already been loaded, as is the case for the Link and Link+Bind test. However, the import time of either test is only a three fold speedup over the Vanilla build's import time, which includes reference counting as well as the core loading and binding functionality.

Second, it reveals that the lazy binding mechanism of this system can significantly increase data cache conflict misses. With lazy binding, the runtime has to transfer control to the dynamic linker whenever a function in an external dynamic library is first referenced. The dynamic linker then resolves the address of the external function for this library, updating the PLT for future references. The frequency at which this task is performed is further exasperated by the fact that each dynamic library maintains its own PLT that is not updated by another library's references. The cache miss counts in Table 2 indicate that such runtime binding operations for dynamically linked applications are memory intensive on their own. Clearly, the memory intensive binding operations performed by the dynamic linker increases the eviction rate of reusable computational cache lines. We theorize that the number of data cache misses at visit time is even greater for real HPC Python applications as they could hold more computational data cache lines before being evicted by the runtime binding logic.

4.2 Real Application vs. Pynamic Model

To evaluate Pynamic's ability to model real applications for HPC development tools, we compared the startup performance of the TotalView parallel debugger between a Python-based multiphysics application at the Lawrence Livermore National Laboratory and a representative Pynamic build. Over the last few years, this application has consistently exposed performance problems in several tools, due in large part to its use of an ever increasing number of DLLs, currently more than five hundred; the startup performance of TotalView has been representative of those issues. We first chose a set of parameters to approximate the aggregate total size of the shared libraries. These parameters include the text, data, debug, symbol table, and string table sizes. Our Pynamic build that approximates these parameters of the multiphysics application consists of 280 Python modules and 215 utility libraries, each averaging 1850 functions. As shown in Table 3, the resulting DLLs exhibited similar properties to the actual application.

section	real app	Pynamic
Text	287	665
Data	9	13
Debug	1100	1100
Symbol Table	17	36
String Table	92	348
total	1504	2162

Table 3: Size comparison in megabytes of a real application and its Pynamic model.

Our test was performed running the multiphysics application and its representative Pynamic build at thirty-two MPI tasks. We measured the tool's startup time that is composed of two phases. The first phase measures the time for the debugger to launch a parallel job and attach to all the tasks at their program startup. Thus, it includes the time for the tool to update the link maps and the symbol tables of the parallel processes for all pre-linked dynamic libraries. The second phase startup measures the tool's ability to handle dynamic loading events generated by inital Python *import* calls. The time comparisons for this test can be seen in Table 4. These results include the first invocation, Cold Startup, as well as a subsequent invocation, Warm Startup. The Warm Startup was about twice as fast as the Cold Startup for both Pynamic and the actual application. We speculate that this speedup is due to TotalView's ability to cache the information from the first run and use it for subsequent runs. Overall, these results show that the Pynamic build is accurate enough to capture TotalView's behavior against the actual application.

Cold/Warm	startup metric	real app	Pynamic
Cold Startup	1^{st} phase	5:28	6:39
Cold Startup	2^{nd} phase	3:35	3:21
Cold Startup	total	9:03	10:00
Warm Startup	1^{st} phase	1:39	1:01
Warm Startup	2^{nd} phase	3:34	3:10
Warm Startup	total	5:13	4:11

Table 4: TotalView starup time comparison (*minutes:seconds*) between a real application and its Pynamic model.

5 Conclusion and Future Work

We have presented the need for a Python benchmark in the HPC community as well as Pynamic, our response to this need. Pynamic creates a user specified number of Python callable and pure C dynamically linked libraries that can emulate the signature of Python applications. The resulting benchmark provides significant insight into the performance of large scale system software and development environment tools.

We have several plans to use Pynamic to test additional system characteristics that can be stressed by Python applications. For instance, new and even existing extreme scale systems with hundreds of thousands of compute nodes will present new challenges to the common practice of loading DLLs from an NFS file system. Pynamic will help determine the scalability of this current practice and thus help prepare for future extreme scale systems. We are also interested in examining the scaling characteristics of Pynamic with respect to the number of DLLs as well as the size of the DLLs.

We have envisioned several enhancements that may provide even further insight into the nature of large scientific Python applications. For example, Pynamic currently covers one hundred percent of the functions generated in a Python module, a property that is not exhibited by real codes. Allowing Pynamic to be configured with a specified code coverage would allow us to gain further insight regarding the benefits of linking the DLLs at link time and optimize the *dlopen* options. We also could support varying the generated function bodies to represent the static and runtime properties of real codes more accurately.

References

[1] Tool Interface Standard (TIS) Executable and Linking Format (ELF). http://x86.ddj.com/ftp/manuals/tools/elf.pdf.

[2] N. Adiga and et al. An Overview of the BlueGene/L Supercomputer. In *Proceedings of IEEE/ACM Supercomputing '02*, pages 1–21, Baltimore, MD, 2002.

[3] J. Asbahr. Developing Game Logic: Python on the Sony Playstation 2 and Nintendo GameCube, 2002. Presentation available at http://conferences.oreillynet.com/presentations /os2002/asbahr_jason.zip.

[4] D. Beazley. Simplified Wrapper and Interface Generator. http://www.swig.org.

[5] D. Beazley and P. Lomdahl. High Performance Molecular Dynamics Modeling with SPaSM : Performance and Portability Issues. In *Proceedings of the Workshop on Debugging and Tuning for Parallel Computer Systems*, pages 337–351, St. Petersburg, FL, USA, 1994.

[6] S. Browne, C. Deane, G. Ho, and P. Mucci. PAPI: A Portable Interface to Hardware Performance Counters. In *Proceedings of Department of Defense HPCMP Users Group Conference*, 1999.

[7] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[8] B. Cobb and et al. AIX Linking and Loading Mechanisms, 2001. http://www-1.ibm.com/servers/esdd/pdfs/aix_11.pdf.

[9] U. Drepper. Security Enhancements in Red Hat Enterprise Linux, 2004. http://people.redhat.com/drepper/nonselsec.pdf.

[10] E. Jones, T. Oliphant, P. Peterson, and et al. SciPy: Open Source Scientific Tools for Python, 2001. http://www.scipy.org.

[11] P. Miller. Parallel, Distributed Scripting with Python. In *Proceedings of the 3rd Linux Clusters Institute International Conference on Linux Cluster: The HPC Revolution*, Chatham, MA, USA, 2002. *IEEE* Computer Society Press, Los Alamitos, CA.

[12] P. Peterson. F2PY Fortran to Python interface generator, 2005. http://cens.ioc.ee/projects/f2py2e.

[13] TotalView Technologies. TotalView Debugger. http://www.totalviewtech.com/productsTV.htm.